WORKING PAPER

ALFRED P. SLOAN SCHOOL OF MANAGEMENT

ON THE IMPLEMENTATION OF SOPHISTICATED

INTERACTIVE SYSTEMS

D. N. NESS - C. R. SPRAGUE - G. A. MOULTON

506-71

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

ON THE IMPLEMENTATION OF SOPHISTICATED

INTERACTIVE SYSTEMS

D. N. NESS - C. R. SPRAGUE - G. A. MOULTON

506-71

On the Implementation of Sophisticated
Interactive Systems
David Ness, Christopher R. Sprague and G. Allen Moulton

This paper defines "sophisticated interactive systems." It then considers the characteristics of the computer system necessary to support such a system. Finally it discusses some strategies which we have found useful in developing such systems.

On the Implementation of Sophisticated
Interactive Systems

David Ness, Christopher R. Sprague, and G. Allen Moulton

Over the past several years we have been involved in the development of several interactive systems. Among these are:

1) The OPS-3 system on M.I.T.'s CTSS (Ness with Greenberger, Jones and Morris (1)).

2) The PRISM system on CP-67/CMS (Ness and Moulton with Martin and Green (2,3)).

3) An Interactive Monitor for an IBM 1620 (Ness and Moulton).

4) An Interactive Engineering Information System operating on an IBM 360/50 under OS (Sprague and Moulton with Pugh).

5) An Interactive Marketing System operating on a DEC PDP-10 computer (Sprague and Ness (4,5,6)).

6) An Interactive Budgeting System on M.I.T.'s CTSS (Ness (7)).

7) Several other (smaller)efforts using M.I.T.'s Multics system and General Electric's Time-Sharing System (in Milan, Italy).

Some observations about these experiences have distinct implications for the implementation of other sophisticated interactive systems. We will first describe some of the characteristics of the class of systems that we are concerned with and then discuss the requirements imposed by such systems on the computer environment. Finally we turn to questions specifically related to the strategies of implementation.

Let us begin by describing some of the major characteristics of the systems that are of concern to us here. First we are interested in systems, i.e., groups of programs broadly speaking devoted to a common purpose, but each of which has, in some sense, an independent existence.

Second, the systems we are concerned with are complex because:

1) Interactions among the components are sufficient in number and quality.

2) The systems are large enough (tens of thousands of statements) to often require several programmers working independently.

3) There is often a necessity of managing a large data base.

We note, here, that having several programmers working on the project introduces a new dimension of complexity, namely it requires that the development process itself be managed (either formally or informally).

So far all of the things mentioned apply equally well to both interactive and batch-processing systems. Let us now define what we mean by interactive.

An interactive system provides quick feedback after performing some (hopefully useful) task or sub-task as commanded by its user. Such systems are typically designed for direct use (e.g., by a manager himself) without technical or clerical intermediaries. Since such users value their time highly, it is important not only that the system provides a quick response, but that the user is able to express his commands easily and efficiently. This implies a convenient problem-oriented interface, which in turn makes the systems designer's problem more complex. It is often important, for example, to be able to recognize rather complex patterns which parallel the user's thought processes.

Since these systems are directly used another problem arises. They are likely (if successful) to generate change in the users themselves, and thus create new demands on the system. Thus they must be flexible and open-ended in order to accommodate these changes and grow with the user's own understanding of his problems.

As a final note, it is important to say that we are concerned with systems which will be developed interactively as well as used interactively.

The Computer Environment

We have found the following to be important characteristics of the kind of

computer enviornment required to implement a sophisticated interactive system:

    1) A Machine Language Programming Capability

    2) Higher Level Languages

    3) Good Input/Output Control

    4) Several Editing (and String Processing) Languages

    5) A Machine Language Level Debugging Package with a Higher Level
       Language Interface

    6) The Ability to Selectively Share Information and Programs

    7) The Ability to Share Active Processes

    8) A Means of Absolute and Reliable Communication with Secondary Storage

    9) Alternative Systems

None of the systems we have worked with have provided all of these facilities.
Some of them have provided only a few. While this suggests that it is indeed
possible to implement a sophisticated system without all of these facilities, that
task is made much more difficult when some of the features are wanting. Let us
now consider each of them in detail.

Machine-Language Programming Capability

A machine langauge programming capability (along with good input-output
device control) allows the systems designer and implemented to develop programs
to perform necessary functions which are not covered by standard system software.
Such circumstances often arise in building a complex interactive system. We have
often been faced with the necessity of building a flexible input routine, some
special disk storage strategy handlers, user language compilers, etc. While it
might be possible to implement some of these procedures in higher level languages,
they tend to be machine specific anyway, and often a higher level language version
is less clear than an assembly version. In such circumstances machine langauge
implementation is the most direct route.

The order code of the machine is an important determinant of the difficulty of machine language implementation. From a programmer's standpoint the PDP-10 order code, for example, is superior to the IBM System/360 or GE-645 order code. As an example, the availability of symmetric forms of similar operations (such as increment a counter and jump on any of the eight results of comparing to zero) makes the coding of many common tasks much more natural. Some of the vehemence of people who do not like machine language programming is surely due to the unnecessary complexity of most machine languages.

Higher-Level Languages

It is clear, however, that a substantial proportion of the procedures in most interactive systems are most easily expressed in a higher level language of some sort. The detailed expression required by a machine langauge is inappropriate to many of the problems that we have seen. In such circumstances a higher level language (like FORTRAN, ALGOL, or PL/1) is not only easier and faster to write, but also easier to understand. This means that it is a simpler task to modify and adapt these procedures over time. The possibility of transferring a program to another system also sometimes proves to be of substantial value.

Input/Output Control

The necessity of a good input/output interface was mentioned above in the discussion of machine language. Interactive systems which support management decision making may require a substantial data base. A tailor-made interface with the user is also frequently important. An input-output system which allows the user to select the actual source or destination of logical data streams is also valuable. For example, the teletype can be used as an input device while a program is being developed and debugged but in production the source device can be redesigned as the disk. One may also want to take output which is normally sent

to the teletype, and send it instead to a disk file so that it can be manipulated by another program (or, printed on a line printer).

Editing or String-Processing

A less obvious requirement is for editing and/or string processing languages. It is clear that at least one editor is necessary. Why do we want more than one? First, we recognize in implementing a sophisticated system that there will be at least two classes of users: 1) the system designers, implementors, and administrators; and 2) the system users. In many cases the system users are not, themselves, programmers. If it is going to be necessary for them to prepare and modify files stored on the disk, a simple and straightforward editor is necessary.

At the same time, the system designers, implementors, and administrators (and some of the users) need a powerful editing facility to assist with complex problems which often show up in the course of their activities. The facility of modifying parts of lines or of making a consistent global text rearrangment in a file are common examples of the kind of operation required.

It is also useful, for system development, implementation, and maintenance purposes to be able to write a "program" in the editing language. This essentially turns the editor into a string-processor. The TECO (8) editor on the PDP-10 processes this capability in a rather crude way. The QED (9) editor (implemented on CTSS and MULTICS) is an even better example of this kind of flexibility. The usefulness of a programmable editing facility will be made clear when we discuss program archives below.

Thus we want some sophisticated editing facilities for one class of user, and some easy-to-use facilities for the other class of user. Most interactive computer environments provide a simple editor. It is unusual for a sophisticated editor to be provided.

Machine-Language Debugging

Another important capability involves being able to see exactly the "stage" of the process we are developing at any point in time. This is most easily accomplished by allowing access to the machine at the machine language or, better, the assembler language level. This kind of facility also usually allows interactive modifications of a program. Trivial modifications may be made during a debugging and testing run without necessitating a full cycle of editing, reassembling or recompiling, and reloading. Examples of this kind of facility are FAPDBG(9) on CTSS, DDT(10) on the PDP-10, and the Probe(11) on Multics.

A related capability can allow us to see and interact with programs written in higher level languages. In building a system involving programs written in both higher level and machine language it is sometimes necessary to be able to look intimately at this interaction. Even in some circumstances where only a higher language is being used this kind of facility proves to be valuable. It is possible, for example, to have the compiler for a higher level language produce an assembler-like symbol table. which can be used by a debugging language to give the implementor access to the machine language program. Such simple things as using internal tables which are directly related to the program writer's symbols make a vast difference in this regard.

An example may clarify this point. In the PDP-10 system, DEC chose to have their FORTRAN compiler generate assembly-like code which uses labels of the form nP to identify lines of code generated from source language program statement n. They also use source language names for variables. This makes it quite easy to understand what source statements produced a given set of object statements.

Selective Sharing of Information

Selective sharing of information allows the system builder to differentiate amongst classes of the user population. He may, for example, allow some users

(typically himself) to read and/or modify a given piece of information, while other users (typically customers) can only read it, or even more stringently can only read it by using some program which he makes available. Such facilities are not always required in the initial design and use of an interactive system. Nevertheless, the ability to share, and to control the sharing of, such information often proves to be important in the long run. Consider, for example, the problem of allowing many users to access a single data base with a common set of programs. Incredibly, there are still newly designed commercial time-sharing services where each user would need his own copy of both the data base and the programs, at least while he was an active user of the information.

Closely related to this is the problem of protection. While we would want all users to access a single data base, for example, we could not tolerate malicious or inadvertent modification of either the data base or the common programs. In some cases, theft of data and/or programs must also be prevented. Our key requirement is selectivity, i.e., we wish to be able to give some users some privileges, and to be able to differentiate amongst different classes of users.

It is also useful to be able to protect the user from himself, particularly while a new program is being debugged. Many systems builders live in fear of a program which "runs away" and proceeds to delete all of the currently active files.

Sharing of Active Processes

The same consideration which requires selective sharing of information and programs also argues for sharing of active processes. Here, however, the concern is often even more visible at the outset. In a multiple user interactive system it is likely that several different users will be performing the process at the same point in time. If this is the case, then the opportunity to have these processes share a common copy of the procedure may prove very useful. If we

can divide a process into a sharable (of length S) and a non-sharable part (of length N) then the total memory requirement for q active simultaneous users would be:

$$qN + S$$

instead of

$$q(N + S) = qN + qS$$

which would be the amount required if the program were not separated into sharable and non-sharable segments. In the PDP-10 system, for example, FORTRAN is divided in to 9K sharable and 2K non-sharable. This means that five active FORTRAN compilations require $5 \cdot 2K + 9K = 19K$ instead of $5 \cdot (9K + 2K) = 55K$ words of active memory. This clearly implies a substantial saving for the system, both in terms of active core requirements, and in terms of the amount of memory which must be "swapped" back and forth between core and secondary storage.

Communication with Secondary Storage

A means of absolute and reliable communications with secondary storage (e.g., a "binary" editing facility) is also of great importance. We suggest, below, that heavy use of secondary storage is often very important in making the design effective. Interestingly enough, none of the systems that we have worked with have provided this facility in other than rudimentary fashion.

This means of communication is important for several reasons. First, although virtually all of the computer environments provide the facility to look at the text information stored in a file, they often do things like eliminate from the text any illegal or ill-formed characters. Such characters may be of little interest when looking at the text visually, but of great importance in terms of the effect they have on the operation of a program which processes the text. Thus it may be very important to be able to detect the presence of such characters in the file.

The ability to modify and/or generate such files also proves to be useful. In the process of testing, a piece of the system may be ready for test before the piece which generates its input is working properly. We thus need to be able to modify this kind of information in a simple and direct way. We found this kind of facility important enough to implement our own capability for performing these operations on the 1620 and the PDP-10. CTSS already had a rudimentary version of this kind of capability.

Alternative Suppliers

A final point to mention, in this catalogue of "necessary" features, is the availability of more than one supplier of the computer environment. At the present time the market place of time-sharing suppliers is highly volatile. This means that new suppliers may come along and offer better prices, or old suppliers may go out of business. Also systems may develop software bugs which may require moving from one system to another if they cannot be quickly found and corrected.

Implementation Strategies

Before concentrating on those strategies especially appropriate to the implementation of interactive systems, we should mention some strategies conventionally pursued in batch environments and sitll valid in interactive situations.

Higher-Level Languages Preferred

As much code as possible should be written in higher-level languages such as FORTRAN, PL/1, ALGOL, etc. Usually, this means faster, easier coding; it almost always means code that is easier to read, document, modify, and understand. A FORTRAN program for example, almost provides its own flowchart. There have been higher-level languages in the past which were hard to read and understand (IPL-V springs to mind), but most have vanished under the pressure of competitors with

some self-documenting features.

Of course, we sometimes choose assembly-level languages for parts of systems. As far as possible these should be kept to small closed subroutines whose functions are microscopic and easily defined. When this is impossible, we find it useful to first write in a higher-level language, and only after the system is checked out, convert selected subroutines to assembly-level code by hand, retaining the higher-level copies as partial documentation.

Know the Machine

Although we strongly favor coding in high-level languages, we constantly find ourselves debugging at the assembly language, or worse yet, machine-language level. There are many reasons for this: disastrous errors often result in no more than an absolute (or at best symbolic) dump; seemingly valid statements in a higher-level language often produce what appears to be strange code, because of implementation restrictions, compiler idiosyncrasies, and the like; and miunderstandings about the interface between the user program and operating system which can often be resolved only at the machine-language level.

All this says that the programmers must be capable of dealing with the machine at the lowest possible level. Some of our most mysterious bugs have hatched out of a failure on this point. For example, we once had a situation where some FORTRAN input/output failed in some program spheres while it always succeeded in others. We developed at least a dozen hypotheses to explain this, but the trivial truth was long in coming - FORTRAN main programs call a set-up entry in the operating system, while our assembly-level main program did not. All our problems came in phases with assembly-level main programs. We would still be wondering about this problem if we had not been willing to debug at the assembly-language level.

Learning the Machine

One strategy for learning the machine has impressed us as especially powerful. Early in development, define several subroutines which are expected to be used often. Parcel these out to the programmers, to be coded in assembly language. This will be a painful process, but everyone will learn the machine. The odds are good that few if any of these subroutines will survive to the final system, but they were not written to survive - their purpose was education.

Keep Statistics

It is easy to overlook the compilation of statistics about system operation. Neglecting this portion of the system, however, means that locating bottlenecks and opportunities for improvement is at best a matter of guesswork. If we wish to follow the strategy of using higher-level language initially and later re-coding in assembly-language, information about the costs of each small part of the system is vital. There are other reasons for collecting such statistics, of course, but the largest payoff is in locating areas of improvement.

Maintain Generality

We find it useful to program in as much generality as possible. For example, the use of variable names and associated DATA statements (in FORTRAN) as a replacement for constants is very handy if a "constant" should change.

A far less trivial point is the use of such "list-processing" techniques as threaded lists, free-space handlers, and arrays of pointers, even in those situations where array sizes and actual core size available are fixed. For one thing, this may prevent extensive re-coding if a new algorithm or new class of problems imposes a major change in the mix of data required. For another, if the program is later moved to another system where the core bound is indeed adjustable, then care exercised in the initial stages will result in greater flexibility and, often

enough, reduced costs. (For some of the dangers of pushing this argument too far see Martin and Moses (12).)

Deletion is Easier Than Insertion

Our final point in this section is that it is far easier to delete statements from a program than to add them. Therefore, we try to build our tracing and debugging statements into our program initially, and remove them later. This also gives us a good excuse for putting useful comments into the code early. When in doubt, put in a trace statement and a comment about it. Final results justify the small extra effort.

We now come to those strategies especially useful when implementing complex interactive systems.

Intermediate Files

We find it a useful practice to pass information from routine to routine by means of intermediate files held in secondary storage called for above, helps in at least two ways:

First, a calamitous failure of the program can be most easily diagnosed if the outputs of and inputs to the various routines can be examined after the fact. When such information is kept in core, it is all too often lost on program failure.

Second, the intermediate files provide a convenient structuring mechanism for the system designers. At an abstract level, the writing of such a file can mark the logical end of a portion of the program - providing a basis for segmenting one's thinking. More practically, actual programming tasks can be easily separated into, say, preparation and use of such a file.

Text Intercommunication Where Possible

If intermediate files are used, and to a lesser extent even if they are not,

it is wise to pass intermediate information as text wherever the volume (and, implicitly, the amount of effort required) is low enough to permit it. The costs are not insignificant: both coding time and running time are required for formatting and translation. The benefits come primarily in the debugging stages, where the difference between information passes as text (and therefore readable) and information passed in internal format can save literally hours per bug. Many errors which are very difficult to detect when only internal data formats are used become obvious when the same data is passed as text.

When the volume of data is large enough to preclude text intercommunication in the final system, it is often still useful to write and check out with text, converting when the operation of each routine seems correct.

Archiving of Information

We usually try to keep closely-related source programs packed together into single files in secondary storage. (Many time-sharing systems provide a facility for combining and separating individual text files. When one is lacking, we typically write one.) Not only does this strategy result in more compact files, it also makes it easy to find a specific source program, and it provides a sort of automatic check-list of programs which need to be examined before a change is made to any of them.

When a program is to be modified, a copy can be extracted from the archive and worked on, while the definitive copy remains safe and untouched in the archive. When modifications are checked out and complete, the old version can be moved to an archive of obsolete code, while the new version replaces it in the definitive archive. This strategy is especially appropriate where there are several programmers working at once. While there is some cost associated with combining and extracting files, it is insignificant compared to the cost resulting from one programmer wiping out another's changes.

Phase Structure

It is often appropriate to divide a system into completely separate phases, each of which monopolizes (virtual) core storage for a time and then terminates by directing the time-sharing system to initiate the next phase. Usually, such a structure is required because of resource limitations or by the economics of the situation. But even when there is no requirement for phase structure, it can be very useful, when combined with communication by intermediate files.

When an error is traced to a specific phase, it is relatively easier to find and correct than if all code was potentially suspect. Even better, the use of hand-built test files allows one to build, check out, and debug phases one by one. They need not even bebuilt in "order".

As the system develops, it is very easy to insert a new phase between two existing phases, to eliminate a phase, or to alter the order in which phases are executed. Another powerful system-development tool which we will call the "alternate-phase feature" can best be described by an example. Suppose there are multiple users B,C,D ... of a system which is contained in "user" area A. Each user has a small initiator program which directs the time-sharing system to execute a program named PHASE1 in area A. When PHASE1 terminates, it directs the time-sharing system to execute PHASE2 in area A, and so on through PHASE9. Each user B,C,D ... has access to the entire system, but none has a copy.

Now suppose that user B is a programmer who is building a new and improved version of PHASE2. Under normal circumstances, he will need a local copy of the entire system, so as not to disturb other users. But, if all phases are programmed to look for their successors first in the local user area (whatever it is) and only on failure to look in area A, then all he needs is his own local version of PHASE2. The initiator program will start the executor of PHASE1 from A. When PHASE1 terminates, it will look for PHASE2 first in the local area (B). Finding one, it will execute it, bypassing the (old) version in A. If PHASE2 terminates normally, it

will look for PHASE3 in area B. Not finding one, it will execute PHASE3 from A.

In this way, any user can have a private version of any phase without disturbing other users and without the necessity for multiple copies of the entire system. In this way new versions of the system can be extensively tested in context before being "published" to the user community.

Direct Posting

In a multiple-phase system with communication by intermediate files there are often alternative communication paths between phases. For example, if PHASE1 produces some data for PHASE2 and some for PHASE3, one can either write two intermediate files, one for each "destination" or alternatively a single file for PHASE2, giving it the responsibility of passing along the data to PHASE3.

Of the two strategies, we vastly prefer the former (direct posting), even though it can result in a large number of intermediate files. The reason is simple: unless we are very careful, the latter strategy (pass-along) means that any modification to the output of PHASE1 destined for PHASE3 requires modifying both PHASE2 and PHASE3. Direct posting assumes that only the directly affected PHASE3 need be modified.

On-Line Documentation

We have recently begun to experiment with interactive approaches to the documentation of systems. One very useful feature is a program which will read a source file, display it line by line, and permit the insertion of comments. It is often easier and useful to other readers, if comments are added after a program is complete, rather than while it is being written.

Another approach involves a combination of three components: First, a guided input program which serves as a checklist of items to be included in a

short document about a program (e.g., calling sequences, parameter descriptions, functional descriptions, etc.); Second, an archive facility for managing documentation files so produced; and third, a printing facility to format such files for easy reading and use.

Although our experience is still relatively sparse, the only obvious disadvantages of this approach are cost and the hazards of losing documentation in process if the system fails.* There are several advantages. Documentation can be done from the same console and about the same time as the programming itself. Documentation files are machine-readable and can be sorted and selected by any desired key. Last but not least, documentation is immediately available, system-wide, once typed in.

Some Comments on Programmers Themselves

While we have learned something about the hardware and software requirements for developing large interactive systems, we have also learned a good deal about the desired characteristics of the people involved in such a development effort.

First, the combination of intelligence and understanding of computation is far more important than experience with a specific machine (or language). It is far better to pay the set-up cost of learning new material than to be saddled with mediocre team members.

Second, the members of a development team should have diverse experience and styles. The point here is that the wider the range of alternatives proposed for any given problem, the better the final solution is likely to be. This does not mean that development should be allowed to proceed helter-skelter, but that more

---

*So far, these costs have not proved severe. It costs us about 10¢ worth of computer time to document an average (50 line) program. Losses due to system failure have been non-existent.

than one point of view should be heard on nearly every question. One useful mech-

anism for communicating differences of approach is to force team members to debug

each other's code. One quickly learns the strengths and weaknesses of one's col-

leagues' styles. It also encourages well-annotated code.

The question of diversity of style can be tackled at a higher level: some of

our most successful collaborations have arisen from the tension between a team

member who wants to "do it right" and one who wants to "do it on time." Of course,

the tension can produce results that are neither right nor on time, but by and

large we find it useful to have one team member take on the role of the "idealist"

and another the "cynic."

Finally, it is not particularly necessary for team members to like one another,

but a good, healthy respect is absolutely indispensible. Blind faith, while touching,

is disastrous. What is required is the constant expectation that the other members

are capable of good work and are trying to deliver it, together with a willing-

ness to be very frank with criticism when work is late or bad.

BIBLIOGRAPHY

1.   Greenberger, M., M. M. Jones, J. H. Morris, and D. N. Ness,
      On-Line Computation and Simulation: The OPS-3 System, MIT
      Press, 1965.

2.   Ness, D. N., W. A. Martin, R. S. Green, and G. A. Moulton,
      The Prism Primer.

3.   Ness, D. N., R. S. Green, W. A. Martin, and G. A. Moulton,
      "Computer Education in a Graduate School of Management,"
      Communications of the ACM, Vol. 13, No. 2, February 1970.

4.   Ness, D. N., and C. R. Sprague, "A Flexible System to Retrieve,
      Manipulate and Display Information from a Stable, Questionnaire
      Oriented Data Base," Part I. Forthcoming.

5.   Sprague, C. R., and D. N. Ness, "A Flexible System to Retrieve,
      Manipulate and Display Information from a Stable, Questionnaire
      Oriented Data Base," Part II.

6.   _____, "A Flexible System to Retrieve, Manipulate and
      Display Information from a Stable, Questionnaire Oriented Data
      Base," Part III.

7.   Ness, D. N., "Interactive Budgeting Models: An Example," MIT
      Sloan School of Management Working Paper No. 345-68, 1968.

8.   Digital Equipment Corporation, The TECO Text Editor and Corrector.

9.   Crisman, P. A., The CTSS Programmer's Guide, Section AH,3.09.

10.   Digital Equipment Corporation, DDT: The Dynamic Debugging Technique.

11.   The Multiplexed Information and Computing Service: Programmers Manual.

12.   Martin, W. A., and J. Moses, "Why Programmers Make Bad Models of
      the Real World," (to be published).